High-Level Synthesis of Pipelined FSM from Loop Nests

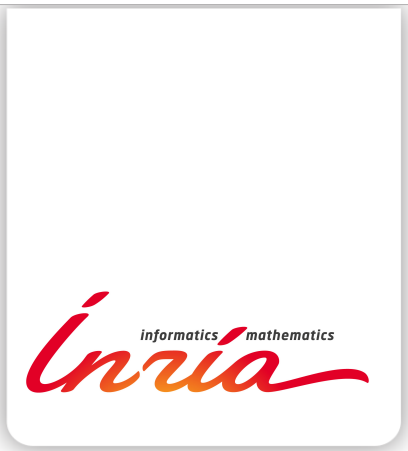
Christophe Alias, Fabrice Rastello, Alexandru Plesco

High-Level Synthesis of Pipelined FSM from Loop Nests

Christophe Alias, Fabrice Rastello, Alexandru Plesco

RESEARCH REPORT
N° 8900
April 2016
Project-Team Roma

ISSN 0249-6399 ISRN INRIA/RR--8900--FR+ENG



High-Level Synthesis of Pipelined FSM from Loop Nests

Christophe Alias*, Fabrice Rastello†, Alexandru Plesco‡

Project-Team Roma

Research Report n° 8900 — April 2016 — 15 pages

Abstract: Embedded systems raise many challenges in power, space and speed efficiency. The current trend is to build heterogeneous systems on a chip with specialized processors and hardware accelerators. Generating an hardware accelerator from a computational kernel requires a deep reorganization of the code and the data. Typically, parallelism and memory bandwidth are met thanks to fine-grain loop transformations. Unfortunately, the resulting control automaton is often very complex and eventually bound the circuit frequency, which limits the benefits of the optimization. This is a major lock, which strongly limits the power of the code optimizations appliable by high-level synthesis tools.
In this report, we propose an architecture of control automaton and an algorithm of high-level synthesis which translates efficiently the control required by fine-grain loop optimizations. Unlike the previous approaches, our control automaton can be pipelined *at will, without any restriction.* Hence, the frequency of the automaton can be as high as possible. Experimental results on FPGA confirms that our control circuit can reach a high frequency with a reasonable resource consumption.

Key-words: High-level synthesis, fine-grain loop optimization, control automaton, pipeline

* Inria/ENS-Lyon/UCBL/CNRS
† Inria
‡ The XtremLogic™ company

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Automate de contrôle pipeliné: architecture et synthèse à partir d'un nid de boucles

Résumé : Les systèmes embarqués soulèvent de nombreux problèmes d'optimisation en termes d'énergie, d'espace et de vitesse d'exécution. La tendance actuelle est de construire des systèmes hétérogènes sur puce avec des processeurs spécialisés et des accélérateurs matériels. Générer un accélérateur matériel à partir d'un noyau de calcul requiert une réorganisation profonde du calcul et des données. Typiquement, le parallélisme et les besoins en bande passante sont obtenus en appliquant des transformations de boucle à grain fin. Malheureusement, l'automate de contrôle résultant est souvent très complexe, au point de limiter la fréquence du circuit. Il s'agit d'un verrou majeur, qui limite fortement la puissance des optimisations applicables par les outils synthèse de circuit haut-niveau.

Dans ce rapport, nous proposons une architecture d'automate de contrôle et un algorithme de synthèse haut-niveau qui génére efficacement le contrôle requis par les transformations de boucle à grain-fin. Contrairement aux approches existantes, notre automate de contrôle peut être pipeliné *à volonté, sans aucune restriction.* Par conséquent, la fréquence de l'automate de contrôle peut être aussi élevée que voulu. Les résultats expérimentaux sur circuit FPGA confirment que notre circuit de contrôle peut atteindre de hautes fréquences, avec une consommation de ressources FPGA raisonnable.

Mots-clés : Synthèse de circuits haut-niveau, transformation de boucles à grain fin, automate de contrôle, pipeline

1 Introduction

Embedded systems raise many challenges in power, space and speed efficiency. The current trend is to build heterogeneous systems on a chip made of specialized processors and dedicated hardware accelerators. Generating an efficient hardware accelerator from a kernel implies a deep reorganization of the code and the data, typically to parallelize the code and to reduce the data transferts. The resulting control automaton may be very complex and eventually bound the circuit frequency. This is a major scientific lock which strongly limits the power of the optimizations appliable by an high-level synthesis tool. The polyhedral model [11] is a mature framework to design powerful automatic parallelizers for loop kernels, as those found in signal processing applications. The outcome is usually a loop transformation specified with a mathematical function. Although the polyhedral model was initially developped to synthesize high-performance circuits [16], the problem of generating an efficient control automaton from a polyhedral transformation is still opened today.

In this report, we propose an architecture of control automaton and an algorithm of high-level synthesis from a polyhedral loop transformation. More precisely:

- We propose an architecture of control automaton which consists of a simple counter connected to a staged scheme in charge of computing the current iteration vector. The stage scheme only requires the counter value to compute the iteration vector. As a consequence, the whole automaton can be pipelined *at will, without any restriction*. This guarantees that the frequency of the automaton can be as high as possible.

- We propose an algorithm of high-level synthesis from a polyhedral loop transformation. Our algorithm acts as any polyhedral code generator. It takes as input the loop nest and the mathematical transformation, then it generates the control automaton.

- The control mainly uses additions and multiplications by a constant. There is a small amount (usually 2 or 3) of full multiplications that are computed only once for the reset phase. The latency is constant, and the throughput is one iteration by cycle.

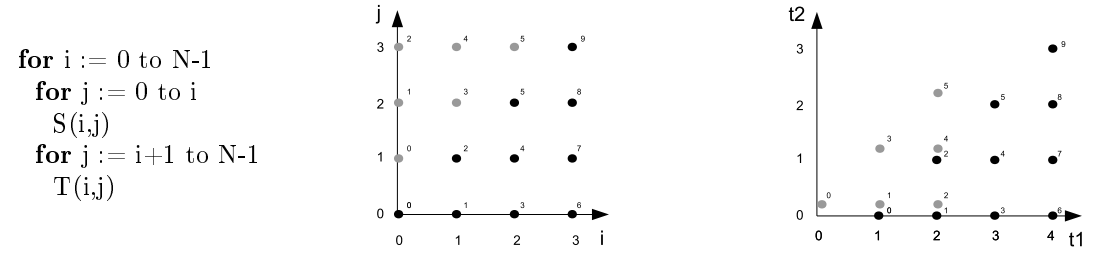
- We have validated our approach experimentally by mapping several loop nests on an Altera FPGA circuit. Our results confirm that the peak frequency can be reached, with, as a bonus, a small resource usage.

The remainder of this report is organized as follows. Section 2 specifies the loop nests considered and polyhedral loop transformations. Section 3 presents the control automaton architecture and the high-level synthesis algorithm. Section 4 discusses possible post-optimizations of the control. Section 5 discusses the related work. Section 6 presents the experimental results. Finally, Section 7 concludes this report and draws some perspectives.

2 Preliminaries

This section specifies the input of our high-level synthesis algorithm. We assume the loops to be affine (next paragraph) and the transformation to be an affine schedule (subsequent paragraph). Finally, we can focus without loss of generality to perfect loop nests (subsequent paragraph).

Affine control In this report, we consider the generation of control circuits for programs with *affine control*. The only control structure allowed are `for` loops and `if`. Moreover, the loop bounds and the condition must be affine functions of the surrounding loop counters and structure



(a) Sample program

(b) Iteration domains $\mathcal{I}_S$ (black) and $\mathcal{I}_T$

(c) Date domains $\mathcal{D}_S$ (black) and $\mathcal{D}_T$ with $\theta_S(i,j) = (i+1, j)$ and $\theta_T(i,j) = (j-1, i)$

Figure 1: Loop nest, iteration domain, transformed domain

parameters. Figure 1.(a) depicts an example of such a program. Note the structure parameter N, typically the size of an array. Also, note that no hypothesis is made on the instruction $S(i,j)$: it could perfectly encapsulate non-affine control, but then its implementation would be left to the user. Affine control allows to represent *exactly* the iterations of loop nests with convex polyhedra. That way, many compiler analysis (dependence, scheduling, allocation, code generation, etc.) can be defined with geometric operations on polyhedra and linear programming [10, 6, 1, 5, 2]. This is the *leitmotiv* of polyhedral compilation, which allows to design precise and powerful program optimizations. On the example, the iterations of $S(i,j)$ can be represented by the mapping $\mathcal{I}_S = N \mapsto \{(i,j) \mid 0 \leq i < N \wedge 0 \leq j \leq i\}$. This mapping is traditionally called the *iteration domain* of S, any $(i,j) \in \mathcal{I}_S$ is called an *iteration vector*. We usually denote by $\langle S, i, j\rangle$ the execution of S at the iteration vector $(i,j) \in \mathcal{I}_S$. Traditionally, S is called a *statement* and $\langle S, i, j\rangle$ is called an *operation*. Figure 1.(b) show the iteration domains of S and T, each iteration vector $x \in \mathcal{I}_s$ (resp. $x \in \mathcal{I}_T$) is labelled by its rank of execution in $\mathcal{I}_S$ (resp. $\mathcal{I}_T$).

Scheduling and control generation Usually, the outcome of a polyhedral optimization is an affine scheduling function $\theta_S : \mathcal{I}_S \to \mathcal{D}$ mapping each iteration vector $x \in \mathcal{I}_S$ of the statement S to a *date* $\theta_S(x) \in \mathcal{D}$. The dates are vectors of integers ordered by the lexicographic order $\mathcal{D} = (\mathbb{N}^p, \ll)$: $(t_1, \ldots, t_n) \ll (u_1, \ldots, u_n)$ iff $t_1 < u_1$, or $t_1 = u_1 \wedge t_2 < u_2$, and so on. In this report, we assume each scheduling function θ_S to be *sequential* (injective): to each date is assigned a unique iteration vector. That way, the scheduling functions are always reversible. Note that it does *not* preclude the global schedule to be parallel: iterations of S and iterations of T could perfectly be executed in parallel. For example, the scheduling function $\theta_S(i,j) = (i+1, j)$ and $\theta_T(i,j) = (j-1, i)$ maps the iterations of S and T to the date domain $\mathcal{D}_S = \theta_S(\mathcal{I}_S)$ and $\mathcal{D}_T = \theta_T(\mathcal{I}_T)$ depicted in figure 1.(c). The iterations are reordered in the lexicographic order of their dates in $\mathcal{D}_S$ and $\mathcal{D}_T$, see the labels with the original rank of execution. In particular, the iterations in the intersection $\mathcal{D}_S \cap \mathcal{D}_T$ are scheduled to be executed in parallel.

Generating a code w.r.t. a schedule amounts to traverse a union of polyhedra ($\mathcal{D}_s$ and $\mathcal{D}_T$) in the lexicographic order. Then, the original iterations are simply recovered by applying the inverse of the schedule (θ_S^{-1} and θ_T^{-1}). Several approaches were proposed to generate a C program given a schedule [5, 7, 4]. The code produced take into account all the corner cases expressed by the schedule. It is usually long, with a lot of control (split `for` loop, `if`), and cannot be applied directly for the purpose of hardware control generation.

Perfect vs non-perfect loop nests A loop nest is said to be *perfect* when each `for` loop contains either a *single* `for` loop or a statement S. This is not the case on that example, as the loop `for i` contains two loops. Usually, a loop nest is split into communicating dataflow processes respecting the Kahn semantics, each process being a basic block (here a statement S) with the surrounding control (`for` loops and `if`). Then, a separate FSM (Finite State Machine) is generated for each process. Finally post-optimization are applied to factorize partially the FSMs. As a consequence, we can restrict our approach to perfect loop nests without loss of generality. In the remainder of this report, we will consider a single perfect loop nest, with an iteration domain $\mathcal{I}$ of dimension n and a schedule $\theta : \mathcal{I} \to \mathcal{D}$.

3 Pipelined FSM

In this section, we describe the main contribution of this report: the architecture of a control automaton and the corresponding high-level synthesis algorithm. The architecture is roughly a counter followed by a staged scheme that computes the iteration vector corresponding to the counter. Section 3.1 defines precisely what we mean by "counter". Then, Section 3.2 shows how the iteration vector can be computed from the counter with a staged scheme. Finally, Section 3.3 refines the staged scheme to remove computation redundancies.

3.1 Ranking functions

Consider an iteration domain $\mathcal{I}$ of dimension n to traverse with respect to an affine schedule θ and the corresponding date domain $\mathcal{D} = \theta(\mathcal{I})$. The *rank* of a date $\vec{t} = (t_1, \dots t_n) \in \mathcal{D}$ is the number of valid dates occurring strictly before $\vec{t}$. The first date has the rank 0, the second date has the rank 1, and so on:

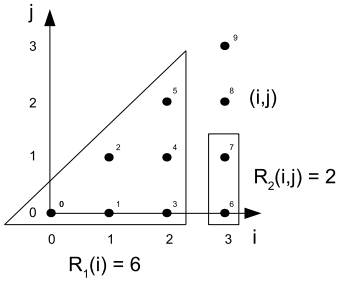
$$\mathbf{R}(\vec{t}) = \text{card } \{\vec{u} \mid \vec{u} \ll \vec{t} \wedge \vec{u} \in \mathcal{D}\} \tag{1}$$

The mapping $\mathbf{R} : \mathcal{D} \to [\![0, \text{card } \mathcal{D} - 1]\!]$ is bijective and stricly increasing with respect to $\ll$: if $\vec{u} \ll \vec{t}$ then $\mathbf{R}(\vec{u}) < \mathbf{R}(\vec{t})$.

The principle of our method is to iterate through the *rank domain* with a simple counter $c \in [\![0, \text{card } \mathcal{D} - 1]\!]$, and then to compute the corresponding date $\vec{t} = \mathbf{R}^{-1}(c)$ and finally the iteration to be executed $\vec{i} = \theta^{-1}(\vec{t})$. Unlike the previous approaches, the computation of $\vec{i} = \theta^{-1}(\mathbf{R}^{-1}(c))$ does not require the previous $\vec{i}$, and can thus be *fully pipelined*, allowing to reach the peak frequency. θ^{-1} is a simple affine transformation $\vec{t} \mapsto A\vec{t} + \vec{b}$. Hence, the remaining of the report will focus on the computation of $\mathbf{R}^{-1}$.

Example Consider the example depicted in Figure 2. We want to build an FSM to traverse the date domain $\mathcal{D} = \{(i, j) \mid 0 \leq i < n \wedge 0 \leq j \leq i\}$. The rank $\mathbf{R}(i, j)$ is the number of points before (i, j) in the lexicographic order. A simple derivation gives $\mathbf{R}(i, j) = i(i+1)/2+j$. There exists an efficient algorithm to count integer points in a convex polyhedron [8], implemented in the Polylib library [13]. Most often, the result is a multivariate polynomial with rational coefficients. In some cases, the result is a piece-wise mapping $\mathbf{R}(\vec{t}) = \vec{t} \in D_1 : P_1(\vec{t}) \dots \vec{t} \in D_q : P_q(\vec{t})$ where each piece $\mathcal{D}_k$ is a convex polyhedron and each expression $P_k(\vec{t})$ is an Ehrhart pseudo-polynomial [8], i.e. a multivariate polynomial whose coefficients change periodically with the value of $\vec{t}$. Observe that our method would accept any computable definition for $\mathbf{R}$ providing $\mathbf{R}$ is positive, bijective and stricly increasing on $\mathcal{D}$.

As we will see in Section 3.2, our algorithm for inverting $\mathbf{R}$, i.e. for finding $(t_1, \dots, t_n)$ given a value of c, iteratively finds t_1, then t_2 (knowing t_1) and so on. To do so, $\mathcal{D}$ is partitioned (into



(a) Date domain $\mathcal{D}$

$$\mathbf{R}(i,j) = i(i+1)/2 + j$$
$$\mathbf{R}_1(i) = i(i+1)/2$$
$$\mathbf{R}_2(i,j) = j$$

(b) Ranking functions $\mathbf{R}$, $\mathbf{R}_1$ and $\mathbf{R}_2$

$$\Delta^1_\ell(i) = 2^\ell.i + 2^{2\ell-1} + 2^{\ell-1}$$
$$\Delta^2_\ell(i,j) = 2^\ell$$

(c) Δ functions

Figure 2: Example of date domain, ranking functions and Δ functions

say $\mathcal{P}_1$, $\mathcal{P}_2$, and so on) thanks to a distribution along the disjunctions of the lexicographic order $\vec{u} \ll \vec{t}$: $\mathcal{P}_1(t_1) = \{\vec{u} \mid u_1 < t_1 \wedge \vec{u} \in \mathcal{D}\}$, $\mathcal{P}_2(t_1,t_2) = \{\vec{u} \mid u_1 = t_1 \wedge u_2 < t_2 \wedge \vec{u} \in \mathcal{D}\}$, etc. Defining $\mathbf{R}_k(t_1,\ldots,t_k) = \text{card } \mathcal{P}_k(t_1,\ldots,t_k)$ the *rank at depth* k, we have:

$$\mathbf{R}(\vec{t}) = \mathbf{R}_1(t_1) + \mathbf{R}_2(t_1,t_2) + \ldots + \mathbf{R}_n(t_1,\ldots,t_n) \tag{2}$$

Intuitively, $\mathbf{R}_1(t_1)$ is the contribution of t_1 to the rank, $\mathbf{R}_1(t_1) + \mathbf{R}_2(t_1,t_2)$ is the contribution of t_2 provided t_1, and so on. Note that each partial sum $\mathbf{R}_1 + \ldots + \mathbf{R}_k$ is defined on the projection of $\mathcal{D}$ across $(t_1,\ldots,t_k)$: $\{(t_1,\ldots,t_k) \mid \vec{t} \in \mathcal{D}\}$, and has the same properties than $\mathbf{R}$: positive, bijective and strictly increasing.

Example (cont'd) We have: $\mathbf{R}_1(i) = \text{card } \{i' \mid i' < i \wedge (i',j') \in \mathcal{D}\}$ (triangle box on Figure 2). Thus $\mathbf{R}_1(i) = i(i+1)/2$. Also, $\mathbf{R}_2(i,j) = \text{card } \{(i',j') \mid i' = i \wedge j' < j \wedge (i',j') \in \mathcal{D}\}$ (rectangle box). Thus: $\mathbf{R}_2(i,j) = j$. We can easily check that $\mathbf{R}(i,j) = \mathbf{R}_1(i) + \mathbf{R}_2(i,j)$.

The next section explains how to compute $\mathbf{R}^{-1}$ by using the $\mathbf{R}_k$.

3.2 Staged computation

Given a rank $c \in [\![0, \text{card } \mathcal{D} - 1]\!]$, the computation of the date $\vec{t} = \mathbf{R}^{-1}(c)$ amounts to find the maximum value:

$$\vec{t^\star} = \max_{\ll}\{\vec{t} \mid \mathbf{R}(\vec{t}) \leq c\}$$

For maximizing $\vec{t} = (t_1,\ldots,t_n)$ with respect to the lexicographic order, we first maximize t_1:

$$t_1^\star = \max\{t_1 \mid \mathbf{R}_1(t_1) \leq c\}$$

Then we maximize t_2 providing $t_1^\star$:

$$t_2^\star = \max\{t_2 \mid \mathbf{R}_1(t_1^\star) + \mathbf{R}_2(t_1^\star,t_2) \leq c\}$$

And so on:

$$\begin{aligned} t_1^\star &= \max\{t_1 \mid \mathbf{R}_1(t_1) \leq c\} \\ t_2^\star &= \max\{t_2 \mid \mathbf{R}_1(t_1^\star) + \mathbf{R}_2(t_1^\star,t_2) \leq c\} \\ &\ldots \\ t_n^\star &= \max\{t_n \mid \mathbf{R}_1(t_1^\star) + \ldots + \mathbf{R}_n(t_1^\star,\ldots,t_n) \leq c\} \end{aligned} \tag{3}$$

This provides a first staged scheme for computing $\mathbf{R}^{-1}(c)$. We will see in the next section how these expressions can be simplified.

Now, let us see how to compute each stage. Let us write $t_1^\star = b_{w-1}^\star.2^{w-1} + \ldots + b_0^\star.2^0$ the binary decomposition of $t_1^\star$ on w bits. Again, as $\mathbf{R}_1$ is strictly increasing, the bits of $t_1^\star$ can be found with a similar staged scheme starting with the most significant bit:

$$b_{w-1}^\star = \max\{b_{w-1} \in \{0,1\} \mid \mathbf{R}_1(b_{w-1}.2^{w-1}) \leq c\} \tag{4}$$

Since b_{w-1} can take only 2 values, Eq. 4 simplifies into:

$$b_{w-1}^\star \equiv \big(\mathbf{R}_1(2^{w-1}) \leq c\big)$$

Similarly, providing $b_{w-1}^\star$, $b_{w-2}^\star$ is computed as follow:

$$b_{w-2}^\star \equiv \big(\mathbf{R}_1(b_{w-1}^\star.2^{w-1} + 2^{w-2}) \leq c\big)$$

And so on:

$$\begin{aligned} b_{w-1}^\star &\equiv \big(\mathbf{R}_1(2^{w-1}) \leq c\big) \\ b_{w-2}^\star &\equiv \big(\mathbf{R}_1(b_{w-1}^\star.2^{w-1} + 2^{w-2}) \leq c\big) \\ &\ldots \\ b_0^\star &\equiv \big(\mathbf{R}_1(b_{w-1}^\star.2^{w-1} + \ldots + b_1^\star.2^1 + 2^0) \leq c\big) \end{aligned} \tag{5}$$

Once all the bits of $t_1^\star$ are known, bits of $t_2^\star$ can be computed in the same way by using to Eq. 3:

$$\begin{aligned} b_{w-1}^\star &\equiv \big(\mathbf{R}_1(t_1^\star) + \mathbf{R}_2(t_1^\star, 2^{w-1}) \leq c\big) \\ b_{w-2}^\star &\equiv \big(\mathbf{R}_1(t_1^\star) + \mathbf{R}_2(t_1^\star, b_{w-1}^\star.2^{w-1} + 2^{w-2}) \leq c\big) \\ &\ldots \\ b_0^\star &\equiv \big(\mathbf{R}_1(t_1^\star) + \mathbf{R}_2(t_1^\star, b_{w-1}^\star.2^{w-1} + \ldots + b_1^\star.2^1 + 2^0) \leq c\big) \end{aligned}$$

This way, each stage of Eq. 3 can be computed with the stages described by Eq. 5. Although this architecture can be fully pipelined and meet the frequency constraints, many redundancies can be removed to keep the resource consumption to a reasonable level. This is the purpose of the next section.

3.3 Removing redundancies

Computing each bit from Equation 5 amounts to evaluate an expression of the form $E = c - \mathbf{R}_1(t^* + 2^\ell)$ where t^* represents the upper $l-1$ already found bits, and 2^l the currently computed one. When $\mathbf{R}_1$ is a multivariate polynomial, it is always possible to develop that expression, and find a multivariate polynomial $\Delta_\ell^1(t^*)$ such that:

$$c - \mathbf{R}_1(t^* + 2^\ell) = c - \mathbf{R}_1(t^*) - \Delta_\ell^1(t^*) \tag{6}$$

This allows to evaluate at a given stage the expression E in terms of its value at the previous stage (say E') by simply subtracting $\Delta_\ell^1(t^*)$ to it: $E := E' - \Delta_\ell^1(t^*)$. The staged scheme (Eq. 5) can then be evaluated with the following algorithm, which eliminates redundancies and reduces the bit-width of intermediate expressions at the same time:

$E' := c - \mathbf{R}_1(0)$
$t_1 := 0$

//Stage 1: bit b_{w-1}
if $(E := E' - \Delta^1_{w-1}(t_1) \geq 0)$ **then**
  $E' := E$
  $t_1 := t_1 + 2^{w-1}$
end if

//Stage 2: bit b_{w-2}
if $(E := E' - \Delta^1_{w-2}(t_1) \geq 0)$ **then**
  $E' := E$
  $t_1 := t_1 + 2^{w-2}$
end if

...
//Stage w: bit b_0
if $(E := E' - \Delta^1_0(t_1) \geq 0)$ **then**
  $E' := E$
  $t_1 := t_1 + 2^0$
end if

The scheme to evaluate t_2 is similar, using $\Delta^2_\ell(t_1, t_2)$ instead of $\Delta^1_\ell(t_1)$ and $\mathbf{R}_2(t_1, 0)$ instead of $\mathbf{R}_1(0)$.

Example (cont'd) We have $\mathbf{R}_1(i + 2^\ell) = (i + 2^\ell)(i + 2^\ell + 1)/2 = R_1(i) + 2^\ell.i + 2^{2\ell-1} + 2^{\ell-1}$, hence $\Delta^1_\ell(i) = 2^\ell.i + 2^{2\ell-1} + 2^{\ell-1}$. Also: $\mathbf{R}_2(i, j + 2^\ell) = j + 2^\ell$, hence $\Delta^2_\ell(i, j) = 2^\ell$. Assuming i and j to be encoded on $w = 2$ bits, Figure 3 depicts the obtained staged scheme to compute $(i, j) = R^{-1}(c)$.

Note that the two last stages of the FSM amount to take directly $j := E'$. This is one of the possible post-optimizations described in the next section to reduce the number of stages of the FSM.

4 Post-optimization

As pointed out earlier, despite the incremental computation of E thanks to the use of Δ, the proposed scheme may contain some redundancies. Also several bitwise computations involving seperate bits may sometimes be combined. With the overall goal of reducing the complexity (size and length) of our circuit we propose to apply standard compiler analysis and optimizations on our scheme. We would like to outline that the experimental results presented further do not use the post-optimization phase.

Bit-width Analysis Bitwidth analysis [17] is a static analysis which goal is to compute the range of bits used by a given code instruction in a program. In the general case this is done using a simple data-flow analysis. In our case we have the ability to compute the values directly by reasoning on the volume of the polyhedrons that the variables represent. To start with detailing this phase, let us denote by $\|v\|$ the range of bits occupied by variable v. By occupied, we mean the bits required to store v i.e. all the bits of v that could be non-zero. As an example, the initial

```
architecture R^{-1}(c: in integer; i, j: out integer)

E' := c - 0 //As R_1(0) = 0
i := 0

//Stage 1: bit b_1 of i
if (E := E' - (2i + 3) ≥ 0) then
   E' := E
   i := i + 2
end if

//Stage 2: bit b_0 of i
if(E := E' - (i + 1) ≥ 0) then
   E' := E
   i := i + 1
end if

E' := E' - 0 //As R_2(i, 0) = 0
j := 0

//Stage 3: bit b_1 of j
if(E := E' - 2 ≥ 0) then
   E' := E
   j := j + 2
end if

//Stage 4: bit b_0 of j
if(E := E' - 1 ≥ 0) then
   E' := E
   j := j + 1
end if
```

Figure 3: Staged scheme for computing $(i, j) = \mathbf{R}^{-1}(c)$ on our running example

value of c in the example of Figure 3 is bounded by the volume of the full triangle i.e. 10 (i and j are both bounded by 3 in our running example). So $\|c\| = [\![0, 3]\!]$: the first 4 bits are required.

We first express the the bit-widths of t_1^*, t_2^*, etc. that are known by construction: at every stage one additional bit is conditionally set. Going back to our example: (1) prior to the first stage $\|i\| = \emptyset$; (2) right after the first stage $\|i\| = [\![1, 1]\!]$; (3) right after the second stage $\|i\| = [\![0, 1]\!]$; etc.

Similarly we can compute the bit-width and the range of the remaining bits to be set: $(t_1 - t_1^*)$, $(t_2 - t_2^*)$, etc. This allows to compute the range of E' as it corresponds to the volume of the remaining polyhedron. As an example, we get that: (1) prior to the first stage $\|E'\| = [\![0, 3]\!]$; (2) right after the first stage $\|E'\| = [\![0, 2]\!]$; (3) right after the second $\|E'\| = [\![0, 1]\!]$, etc.

Bit-shrink Once we computed the bit-width of all variables and expressions, this allows to compute the bits used by a given operation. By *used* we consider different meanings depending

if we are referring to an assignment or to a comparison. Hence, for operation `j+=2`[1], if we know for example that $\|j\|_{max} < 1$ then we know that the corresponding sets bit-1 to 1 i.e. "uses" only bit-1. Otherwise, we consider that it "uses" all the bits of $\|j\| + 1$. For a comparison operation, *used bits* refer to the actual set of bits that needs to be considered. Hence for $E' \geq 2$, knowing that $\|E'\|_{max} \leq 1$ allows to conclude that it is equivalent to checking if the bit-1 of E' is bigger than 1 which by code re-selection simplifies in checking if bit-1 of E' is equal to 1.

Going back to our running example, bit-shrinking would lead to the following specializations: (1) the first-stage comparison is equivalent to checking whether $E' - 3 \geq 0$; if the condition is met the assignment to i corresponds to set bit-1 to 1; (2) The third-stage comparison is equivalent in checking whether bit-1 of E' is equal to 1 (as explained above); if the condition is met then the assignment to j corresponds to set its bit-1 to 1.

If-conversion If conversion corresponds to replace a control dependence by some predicated instructions. In C syntax this can be viewed as replacing `if(c){v=E}` by `v=c?E:c`. This is especially interesting in our case when the conditionally executed instruction is an assignment. In particular we are interested in a special where code re-selection allows to get rid of the predicate.

Going back to our running example, this turns out to be the case for the last two stages: (1) Recall that the third stage is equivalent to set bit-1 of j to 1 if the bit-1 of E' is 1. In other words this is equivalent to setting bit-1 of j to the value of bit-1 of E' (recall that we know that the bit is originally not used by j).

Vectorization Bit-width analysis allows to refine dependence information. Indeed raw analysis leads to taking the def-use chains in the program as the set of dependencies. Obviously whenever the bit-used by two different operations are not overlapping, one can consider there is no dependencies.

Going back again to our running example this turns out to be the case between stage 3 and stage 4. Indeed stage 3 uses bit-1 of both E' and j while stage 4 uses bit-0 of both E' and j. The consequence is that those two operations can be done in parallel and last two stages can be simplified into `j:=E'; E':=0;`.

5 Related work

In this section we compare our work with previous approaches to generate efficient implementation of FSMs for loop nests in HLS tools. We present the novelty of our approach and continue with the comparisons with existing approaches in the experimental validation section.

In a seminal paper, Boulet *et al.* [7] propose to generate directly a control automaton with polyhedral techniques. The initial motivation was not HLS, but software optimization. This said, this approach contains ideas which inspired HLS specific approaches, as [19], described in the next paragraph. Given a affine loop nest and an affine schedule θ, they derive directly a function $First()$, returning the first iteration vector to be executed ; and a function Next($\vec{x}$), which computes the next iteration to be executed, provided the current iteration $\vec{x}$. These functions are provided as piece-wise affine functions. Unfortunately, the number of pieces increase exponentially with the dimension of the polyhedron. Despite the possible simplifications, the size usually explodes, as well as the critical path, leading to low frequencies.

Yuki *et al.* [19] propose an adaptation of Boulet's technique with the ability to pipeline the control. Boulet's technique exhibits a direct data dependence from $\vec{x}$ to Next($\vec{x}$), then it cannot be pipelined directly. Yuki *et al* propose to increase the dependence distance by composing the

Next function: the recurrence becomes $\vec{x}_{t+d} = \text{Next}^d(\vec{x}_t)$. That way, the Next function can be pipelined with d stages, allowing to increase the frequency of the circuit. Unfortunately, Next^d increases exponentially with d, which limits the possible pipelined depth d and then the maximum frequency reachable by this technique.

Pouchet *et al.* [20] propose to adapt code generation techniques usually applied for software parallelization [5], in order to increase the efficiency of the loop control hardware. However, the problem remains partially open: as even for the simple GEMM algorithm the efficiency of loops presented in their aproach is 7 times smaller than the optimal number of cycles deduced from [14] and the operating frequencies remains very low for an high-end FPGA in the vecinity of 100 MHz. Loop tiling is applied on 20x20x5 size loops and the inner loops are software pipelined by Vivado HLS tool [18]. The authors use rectangular tile overapproximation using a bounding box and a filtering of real iterations. This increase significantly the operating frequency, but at the price of reducing the throughput. When the schedule assume a throughput strictly equal to one, as for example in [3], this would not be suitable. Software pipelining techniques [12] are known to be very inefficient when applied on loops nests with small iteration counts. However, tile size determines the ratio of memory to computing ressources used on FPGA, and bigger tile size increase the size of memory buffers.

6 Experimental results

In this section we present the experimental results of our approach applied on a few examples: iteration over a rectangular and triangular domain in 2 and 3 dimensions with 8 bits unsigned domains in each dimension.

6.1 Experimental setup

We have generated the control automata for the following *date* domains, parametrized with integers N, P and Q:

Rectangle 2D: $\mathcal{D} = \{(i,j) \mid 0 \leq i < N \wedge 0 \leq j < P\}$
Triangle 2D: $\mathcal{D} = \{(i,j) \mid 0 \leq i < N \wedge 0 \leq j \leq i\}$
Rectangle 3D: $\mathcal{D} = \{(i,j,k) \mid 0 \leq i < N \wedge 0 \leq j < P \wedge 0 \leq k < Q\}$
Triangle 3D: $\mathcal{D} = \{(i,j,k) \mid 0 \leq i < N \wedge 0 \leq j < P \wedge 0 \leq k \leq i\}$

The same domains were used in [19], to which we compare our results. For synthesis we used the Quartus Prime Standard [15] from Altera, performed function level verification using ghdl, the clock was connected to global clock routing ressources and the rest of the I/Os were set as virtual. The frequency estimation was done using 85C slow model. Table 1 summarizes the synthesis results obtained.

6.2 Discussion

Execution frequency We have obtained higher frequency than the other approach in all the examples. However, the pipelining frequency can be improved even further. The TimeQuest timing analysis reported critical paths in between stages that have multiple arithmetic operations. The Quartus compiler is unable to correctly retime the design. Significant frequency gains can be obtained by using dedicate pipelined arithmetic operators from for example FloPoCo library [9]. High bit counter source alone can run at a much higher frequency thanks to dedicated high speed arithmetic carry wires between ALMs. We connected our clock to global clock distribution. The frequency of our results increase by about 10% - 20% when connected to virtual pins (for example in case of clock gating with an enable).

Table 1: Experimental results

Benchmark	Version	FPGA	ALUT	3iALUT	FF	DSP	Frequency (MHz)	Loop efficiency %
Rectangle 2D	[19]	StratixIV	201	-	182	-	454	-
	ours	StratixIV	204	191	360	1	511	100
		StratixV	205	192	360	1	595	100
		Arria10	207	191	360	1	623	100
Rectangle 3D	[19]	StratixIV	348	-	215	-	387	
	ours	StratixIV	512	487	880	3	414	100
		StratixV	515	489	880	2	549	100
		Arria10	515	491	880	2	517	100
Triangle 2D	[19]	StratixIV	131	-	111	-	433	-
	ours	StratixIV	279	266	409	1	443	100
		StratixV	279	266	409	1	540	100
		Arria10	270	257	409	1	572	100
Triangle 3D	[19]	StratixIV	384	-	290	-	346	-
	ours	StratixIV	739	718	1287	3	350	100
		StratixV	738	717	1287	2	403	100
		Arria10	730	709	1287	2	414	100

Logic utilization and routing We cannot really compare our results in terms of ALUT with the other approach as it is not mentioned what input size of ALUT their designs is using. An ALM can contain up to 4 LUT if size is smaller than 3 and up to 1 if input size is greater than 6. Our design actualy performed better when Quartus is configured to optimized for minimum size in terms of frequency and ALM compaction than when it is instructed to optimize for performance, thus confirming the limitations of the compiler's retiming algorithm. Most of our ALUTs are of three or less than three input size. This can help tremedously with the integration of our FSMs in a tightly packed FPGA as routing constraints are much lower than when 6 or 7 input ALUTs are used. Our designs have very low DSP usage, and thus can placed and routed easily. In the other works, the DSP usage is not presented.

In our approach the ressource usage represents a very small fraction of big floating poing datapaths like ones found in algorithms from PolyBench benchmark, and therefore overall performance gains justify potentially higher ressource usage.

7 Conclusion

In this report, we have presented a novel architecture of control automaton and an high-level synthesis algorithm from transformed affine loop nests. Unlike previous approaches, our control automation can be pipelined *at will*. As a consequence, it can reach any frequency, and will never limit the frequency of the circuit, whatever the complexity of the loop transformation.

This opens many perspectives in high-level synthesis of hardware accelerators. In particular, our method makes it possible to apply the powerful polyhedral loop transformations in the context of circuit synthesis. In the future, we want to investiguate the application of this method for non-linear loop transformations as loop tiling. Though our method can already be applied to control the execution of a single tile, several issues must be solved to generate the whole control automaton.

References

[1] Christophe Alias, Fabrice Baray, and Alain Darte. Bee+Cl@k: An implementation of lattice-based array contraction in the source-to-source translator Rose. In *ACM Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'07)*, 2007.

[2] Christophe Alias, Alain Darte, and Alexandru Plesco. Optimizing remote accesses for offloaded kernels: Application to high-level synthesis for fpga. In *ACM SIGDA Intl. Conference on Design, Automation and Test in Europe (DATE)*, 2013.

[3] Christophe Alias, Bogdan Pasca, and Alexandru Plesco. Fpga-specific synthesis of loop-nests with pipelined computational cores. *Microprocessors and Microsystems - Embedded Hardware Design*, 36(8):606–619, 2012.

[4] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), Williamsburg, Virginia, USA, April 21-24, 1991*, pages 39–50, 1991.

[5] Cédric Bastoul. Efficient code generation for automatic parallelization and optimization. In *2nd International Symposium on Parallel and Distributed Computing (ISPDC 2003), 13-14 October 2003, Ljubljana, Slovenia*, pages 23–30, 2003.

[6] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 101–113, 2008.

[7] Pierre Boulet and Paul Feautrier. Scanning polyhedra without Do-loops. In *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 4–9, 1998.

[8] Philippe Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. In *Proceedings of the 10th international conference on Supercomputing, ICS 1996, Philadelphia, PA, USA, May 25-28, 1996*, pages 278–285, 1996.

[9] Florent de Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with FloPoCo. *IEEE Design & Test of Computers*, 28(4):18–27, July 2011.

[10] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[11] Paul Feautrier and Christian Lengauer. Polyhedron model. In *Encyclopedia of Parallel Computing*, pages 1581–1592. 2011.

[12] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 318–328, New York, NY, USA, 1988. ACM.

[13] Polylib – A library of polyhedral functions. `http://www.irisa.fr/polylib`.

[14] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/˜ pouchet/software/polybench/[cited July,]*, 2012.

[15] Altera Quartus Prime. `https://www.altera.com/products/design-software/fpga-design/quartus-prime/overview.html`.

[16] Patrice Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proceedings of the 11th Annual Symposium on Computer Architecture, Ann Arbor, USA, June 1984*, pages 208–214, 1984.

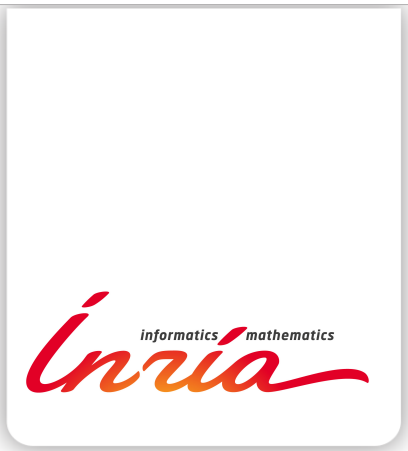
[17] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bidwidth analysis with application to silicon compilation. In *ACM SIGPLAN Notices*, volume 35, pages 108–120. ACM, 2000.

[18] Xilinx Vivado HLS. `http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`.

[19] Tomofumi Yuki, Antoine Morvan, and Steven Derrien. Derivation of efficient FSM from loop nests. In *2013 International Conference on Field-Programmable Technology, FPT 2013, Kyoto, Japan, December 9-11, 2013*, pages 286–293, 2013.

[20] Wei Zuo, Peng Li, Deming Chen, Louis-Noël Pouchet, Shunan Zhong, and Jason Cong. Improving polyhedral code generation for high-level synthesis. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '13, pages 15:1–15:10, Piscataway, NJ, USA, 2013. IEEE Press.

Contents



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399